6.824 - Spring 2022

6.824 Lab 1: MapReduce

Due: Friday Feb 11 23:59ET (MIT Time)

Collaboration policy // Submit lab // Setup Go // Guidance // Piazza

---

Introduction

In this lab you'll build a MapReduce system. You'll implement a worker process that calls application Map and Reduce functions and handles reading and writing files, and a coordinator process that hands out tasks to workers and copes with failed workers. You'll be building something similar to the MapReduce paper. (Note: this lab uses "coordinator" instead of the paper's "master".)

Getting started

You need to setup Go to do the labs.

Fetch the initial lab software with git (a version control system). To learn more about git, look at the Pro Git book or the git user's manual.

```
$ git clone git://g.csail.mit.edu/6.824-golabs-2022 6.824
$ cd 6.824
$ ls
Makefile src
$
```

We supply you with a simple sequential mapreduce implementation in `src/main/mrsequential.go`. It runs the maps and reduces one at a time, in a single process. We also provide you with a couple of MapReduce applications: word-count in `mrapps/wc.go`, and a text indexer in `mrapps/indexer.go`. You can run word count sequentially as follows:

```
$ cd ~/6.824
$ cd src/main
$ go build -race -buildmode=plugin ../mrapps/wc.go
$ rm mr-out*
$ go run -race mrsequential.go wc.so pg*.txt
$ more mr-out-0
A 509
ABOUT 2
ACT 8
...
```

(Note: the **-race** enables the Go race detector. We suggest you develop and test your 6.824 lab code with the race detector. When we grade your labs, we will **not** use the race detector. Nevertheless, if your code has races, there's a good chance it will fail when we test it even without the race detector.)

`mrsequential.go` leaves its output in the file `mr-out-0`. The input is from the text files named `pg-xxx.txt`.

Feel free to borrow code from `mrsequential.go`. You should also have a look at `mrapps/wc.go` to see what MapReduce application code looks like.

Your Job (moderate/hard)

Your job is to implement a distributed MapReduce, consisting of two programs, the coordinator and the worker. There will be just one coordinator process, and one or more worker processes executing in parallel. In a real system the workers would run on a bunch of different machines, but for this lab you'll run them all on a single machine. The workers will talk to the coordinator via RPC. Each worker process will ask the coordinator for a task, read the task's input from one or more files, execute the task, and write the task's output to one or more files. The coordinator should notice if a worker hasn't completed its task in a reasonable amount of time (for this lab, use ten seconds), and give the same task to a different worker.

We have given you a little code to start you off. The "main" routines for the coordinator and worker are in `main/mrcoordinator.go` and `main/mrworker.go`; don't change these files. You should put your implementation in `mr/coordinator.go`, `mr/worker.go`, and `mr/rpc.go`.

Here's how to run your code on the word-count MapReduce application. First, make sure the word-count plugin is freshly built:

```
$ go build -race -buildmode=plugin ../mrapps/wc.go
```

In the `main` directory, run the coordinator.

```
$ rm mr-out*
$ go run -race mrcoordinator.go pg-*.txt
```

The `pg-*.txt` arguments to `mrcoordinator.go` are the input files; each file corresponds to one "split", and is the input to one Map task. The `-race` flags runs go with its race detector.

In one or more other windows, run some workers:

```
$ go run -race mrworker.go wc.so
```

When the workers and coordinator have finished, look at the output in `mr-out-*`. When you've completed the lab, the sorted union of the output files should match the sequential output, like this:

```
$ cat mr-out-* | sort | more
A 509
ABOUT 2
ACT 8
...
```

We supply you with a test script in `main/test-mr.sh`. The tests check that the `wc` and `indexer` MapReduce applications produce the correct output when given the `pg-xxx.txt` files as input. The tests also check that your implementation runs the Map and Reduce tasks in parallel, and that your implementation recovers from workers that crash while running tasks.

If you run the test script now, it will hang because the coordinator never finishes:

```
$ cd ~/6.824/src/main
$ bash test-mr.sh
*** Starting wc test.
```

You can change `ret := false` to true in the Done function in `mr/coordinator.go` so that the coordinator exits immediately. Then:

```
$ bash test-mr.sh
*** Starting wc test.
sort: No such file or directory
cmp: EOF on mr-wc-all
--- wc output is not the same as mr-correct-wc.txt
--- wc test: FAIL
$
```

The test script expects to see output in files named `mr-out-X`, one for each reduce task. The empty implementations of `mr/coordinator.go` and `mr/worker.go` don't produce those files (or do much of anything else), so the test fails.

When you've finished, the test script output should look like this:

```
$ bash test-mr.sh
*** Starting wc test.
--- wc test: PASS
*** Starting indexer test.
--- indexer test: PASS
*** Starting map parallelism test.
--- map parallelism test: PASS
*** Starting reduce parallelism test.
--- reduce parallelism test: PASS
*** Starting crash test.
--- crash test: PASS
*** PASSED ALL TESTS
$
```

You'll also see some errors from the Go RPC package that look like

```
2019/12/16 13:27:09 rpc.Register: method "Done" has 1 input parameters; needs exactly
```

Ignore these messages; registering the coordinator as an RPC server checks if all its methods are suitable for RPCs (have 3 inputs); we know that `Done` is not called via RPC.

A few rules:

- The map phase should divide the intermediate keys into buckets for `nReduce` reduce tasks, where `nReduce` is the number of reduce tasks -- argument that `main/mrcoordinator.go` passes to `MakeCoordinator()`. So, each mapper needs to create `nReduce` intermediate files for consumption by the reduce tasks.
- The worker implementation should put the output of the X'th reduce task in the file `mr-out-X`.
- A `mr-out-X` file should contain one line per Reduce function output. The line should be generated with the Go `"%v %v"` format, called with the key and value. Have a look in `main/mrsequential.go` for the line commented "this is the correct format". The test script will fail if your implementation deviates too much from this format.
- You can modify `mr/worker.go`, `mr/coordinator.go`, and `mr/rpc.go`. You can temporarily modify other files for testing, but make sure your code works with the original versions; we'll test with the original versions.
- The worker should put intermediate Map output in files in the current directory, where your worker can later read them as input to Reduce tasks.
- `main/mrcoordinator.go` expects `mr/coordinator.go` to implement a `Done()` method that returns true when the MapReduce job is completely finished; at that point,

`mrcoordinator.go` will exit.

- When the job is completely finished, the worker processes should exit. A simple way to implement this is to use the return value from `call()`: if the worker fails to contact the coordinator, it can assume that the coordinator has exited because the job is done, and so the worker can terminate too. Depending on your design, you might also find it helpful to have a "please exit" pseudo-task that the coordinator can give to workers.

Hints

- The Guidance page has some tips on developing and debugging.
- One way to get started is to modify `mr/worker.go`'s `Worker()` to send an RPC to the coordinator asking for a task. Then modify the coordinator to respond with the file name of an as-yet-unstarted map task. Then modify the worker to read that file and call the application Map function, as in `mrsequential.go`.
- The application Map and Reduce functions are loaded at run-time using the Go plugin package, from files whose names end in `.so`.
- If you change anything in the `mr/` directory, you will probably have to re-build any MapReduce plugins you use, with something like `go build -race -buildmode=plugin ../mrapps/wc.go`
- This lab relies on the workers sharing a file system. That's straightforward when all workers run on the same machine, but would require a global filesystem like GFS if the workers ran on different machines.
- A reasonable naming convention for intermediate files is `mr-X-Y`, where X is the Map task number, and Y is the reduce task number.
- The worker's map task code will need a way to store intermediate key/value pairs in files in a way that can be correctly read back during reduce tasks. One possibility is to use Go's `encoding/json` package. To write key/value pairs in JSON format to an open file:

  ```
  enc := json.NewEncoder(file)
  for _, kv := ... {
    err := enc.Encode(&kv)
  ```

  and to read such a file back:

  ```
  dec := json.NewDecoder(file)
  for {
    var kv KeyValue
    if err := dec.Decode(&kv); err != nil {
      break
    }
    kva = append(kva, kv)
  }
  ```

- The map part of your worker can use the `ihash(key)` function (in `worker.go`) to pick the reduce task for a given key.
- You can steal some code from `mrsequential.go` for reading Map input files, for sorting intermediate key/value pairs between the Map and Reduce, and for storing Reduce output in files.
- The coordinator, as an RPC server, will be concurrent; don't forget to lock shared data.
- Use Go's race detector, with `go build -race` and `go run -race`. `test-mr.sh` by default runs the tests with the race detector.
- Workers will sometimes need to wait, e.g. reduces can't start until the last map has finished. One possibility is for workers to periodically ask the coordinator for work, sleeping with `time.Sleep()` between each request. Another possibility is for the relevant RPC handler in the coordinator to have a loop that waits, either with `time.Sleep()` or `sync.Cond`. Go runs the handler for each RPC in its own thread, so the fact that one handler is waiting won't prevent the coordinator from processing other RPCs.

- The coordinator can't reliably distinguish between crashed workers, workers that are alive but have stalled for some reason, and workers that are executing but too slowly to be useful. The best you can do is have the coordinator wait for some amount of time, and then give up and re-issue the task to a different worker. For this lab, have the coordinator wait for ten seconds; after that the coordinator should assume the worker has died (of course, it might not have).
- If you choose to implement Backup Tasks (Section 3.6), note that we test that your code doesn't schedule extraneous tasks when workers execute tasks without crashing. Backup tasks should only be scheduled after some relatively long period of time (e.g., 10s).
- To test crash recovery, you can use the `mrapps/crash.go` application plugin. It randomly exits in the Map and Reduce functions.
- To ensure that nobody observes partially written files in the presence of crashes, the MapReduce paper mentions the trick of using a temporary file and atomically renaming it once it is completely written. You can use `ioutil.TempFile` to create a temporary file and `os.Rename` to atomically rename it.
- `test-mr.sh` runs all its processes in the sub-directory `mr-tmp`, so if something goes wrong and you want to look at intermediate or output files, look there. You can temporarily modify `test-mr.sh` to `exit` after the failing test, so the script does not continue testing (and overwrite the output files).
- `test-mr-many.sh` provides a bare-bones script for running `test-mr.sh` with a timeout (which is how we'll test your code). It takes as an argument the number of times to run the tests. You should not run several `test-mr.sh` instances in parallel because the coordinator will reuse the same socket, causing conflicts.
- Go RPC sends only struct fields whose names start with capital letters. Sub-structures must also have capitalized field names.
- When passing a pointer to a reply struct to the RPC system, the object that `*reply` points to should be zero-allocated. The code for RPC calls should always look like

  ```
  reply := SomeType{}
  call(..., &reply)
  ```

  without setting any fields of reply before the call. If you don't follow this requirement, there will be a problem when you pre-initialize a reply field to the non-default value for that datatype, and the server on which the RPC executes sets that reply field to the default value; you will observe that the write doesn欽檛 appear to take effect, and that on the caller side, the non-default value remains.

No-credit challenge exercises

CHALLENGE

Implement your own MapReduce application (see examples in `mrapps/*`), e.g., Distributed Grep (Section 2.3 of the MapReduce paper).

CHALLENGE

Get your MapReduce coordinator and workers to run on separate machines, as they would in practice. You will need to set up your RPCs to communicate over TCP/IP instead of Unix sockets (see the commented out line in `Coordinator.server()`), and read/write files using a shared file system. For example, you can `ssh` into multiple Athena cluster machines at MIT, which use AFS to share files; or you could rent a couple AWS instances and use S3 for storage.